Alloy Analyzer 4 Tutorial

Session 1: Intro and Logic

Greg Dennis and Rob Seater
Software Design Group, MIT
agenda

- Session 1: Intro & Logic
  - break
- Session 2: Language & Analysis
  - lunch
- Session 3: Static Modeling
  - break
- Session 4: Dynamic Modeling
trans-atlantic analysis

- notation inspired by Z
  - sets and relations
  - uniformity
  - *but* not easily analyzed

- analysis inspired by SMV
  - billions of cases in seconds
  - counterexamples not proofs
  - *but* not declarative
why declarative design?

I conclude there are two ways of constructing a software design.

One way is to make it so simple there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

– Tony Hoare [Turing Award Lecture, 1980]
why automated analysis?

The first principle is that you must not fool yourself, and you are the easiest person to fool.

– Richard P. Feynman
alloy case studies

- Multilevel security (Bolton)
- Multicast key management (Taghdiri)
- Rendezvous (Jazayeri)
- Firewire (Jackson)
- Intentional naming (Khurshid)
- Java views (Waingold)
- Access control (Zao)
- Proton therapy (Seater, Dennis)
- Chord peer-to-peer (Kaashoek)
- Unison file sync (Pierce)
- Telephone switching (Zave)
four key ideas . . .

1) everything is a relation

2) non-specialized logic

3) counterexamples & scope

4) analysis by SAT
1) everything's a relation

- Alloy uses relations for
  - all data types – even sets, scalars, tuples
  - structures in space and time
- key operator is **dot** join
  - relational join
  - field navigation
  - function application
why relations?

- easy to understand
  - binary relation is a graph or mapping
- easy to analyze
  - first order (tractable)
- uniform

\[\text{set of addresses associated with name } n \text{ in set of books } B\]

\begin{align*}
\text{Alloy:} & \quad n.(B.\text{addr}) \\
\text{Z:} & \quad \bigcup \{ b : B \land b.\text{addr} (\mid \{n\} \mid) \} \\
\text{OCL:} & \quad B.\text{addr}[n] \rightarrow \text{asSet()}
\end{align*}

There is no problem in computer science that cannot be solved by an extra level of indirection.

– David Wheeler
2) non-specialized logic

- No special constructs for state machines, traces, synchronization, concurrency . . .
3) counterexamples & scope

- observations about design analysis:
  - most assertions are wrong
  - most flaws have small counterexamples

- testing:
  a few cases of arbitrary size

- scope-complete:
  all cases within a small bound
4) analysis by SAT

- SAT, the quintessential hard problem (Cook 1971)
  - SAT is hard, so reduce SAT to your problem
- SAT, the universal constraint solver (Kautz, Selman, ... 1990's)
  - SAT is easy, so reduce your problem to SAT
  - solvers: Chaff (Malik), Berkmin (Goldberg & Novikov), ...

Stephen Cook  Eugene Goldberg  Henry Kautz  Sharad Malik  Yakov Novikov
Moore's Law
SAT performance
SAT trophies

[Image of four SAT trophies]
install the Alloy Analyzer

➢ Requires Java 5 runtime environment
  – http://java.sun.com/

➢ download the Alloy Analyzer 4

➢ run the Analyzer
  – double click alloy4.jar or
  – execute java -jar alloy4.jar at the command line

➢ this bullet indicates something you should do
verify the installation

➢ Click the “file” menu, then click “open sample models” to open `examples/toys/ceilingsAndFloors.als`

➢ click the “Execute” icon
  - output shows graphic

• need troubleshooting?
modeling “ceilings and floors”

sig Platform { }
    there are “Platform” things

sig Man {ceiling, floor: Platform}
    each Man has a ceiling and a floor Platform

pred Above [m, n: Man] {m.floor = n.ceiling}
    Man m is “above” Man n if m's floor is n's ceiling

fact {all m: Man | some n: Man | Above[n,m] }
    "One Man's Ceiling Is Another Man's Floor"
checking “ceilings and floors”

```
assert BelowToo { 
    all m: Man | some n: Man | Above [m,n]
}

"One Man's Floor Is Another Man's Ceiling"?
```

```
check BelowToo for 2

check "One Man's Floor Is Another Man's Ceiling"

counterexample with 2 or less platforms and men?
```

• clicking “Execute” ran this command
  – counterexample found, shown in graphic
counterexample to “BelowToo”
Alloy = logic + language + analysis

- logic
  - first order logic + relational calculus

- language
  - syntax for structuring specifications in the logic

- analysis
  - bounded exhaustive search for counterexample to a claimed property using SAT
software abstractions
logic: relations of atoms

- atoms are Alloy's primitive entities
  - indivisible, immutable, uninterpreted

- relations associate atoms with one another
  - set of tuples, tuples are sequences of atoms

- every value in Alloy logic is a relation!
  - relations, sets, scalars all the same thing
logic: everything's a relation

- sets are unary (1 column) relations

  Name = \{(N0), (N1), (N2)\}
  Addr = \{(A0), (A1), (A2)\}
  Book = \{(B0), (B1)\}

- scalars are singleton sets

  myName = \{(N1)\}
  yourName = \{(N2)\}
  myBook = \{(B0)\}

- binary relation

  names = \{(B0, N0), (B0, N1), (B1, N2)\}

- ternary relation

  addrs = \{(B0, N0, A0), (B0, N1, A1), (B1, N1, A2), (B1, N2, A2)\}
logic: relations

\[
\text{addr} = \{(B0, N0, A0), (B0, N1, A1), (B1, N1, A2), (B1, N2, A2)\}
\]

- rows are unordered
- columns are ordered but unnamed
- all relations are first-order
  - relations cannot contain relations, no sets of sets
logic: address book example

Name = \{(N0), (N1), (N2)\}
Addr = \{(A0), (A1), (A2)\}
Target = \{(N0), (N1), (N2), (A0), (A1), (A2)\}
address = \{(N0, A1), (N1, N2), (N2, A1), (N2, A0)\}
## logic: constants

<table>
<thead>
<tr>
<th>none</th>
<th>empty set</th>
</tr>
</thead>
<tbody>
<tr>
<td>univ</td>
<td>universal set</td>
</tr>
<tr>
<td>iden</td>
<td>identity relation</td>
</tr>
</tbody>
</table>

Name = {\( (N0), (N1), (N2) \)}
Addr = {\( (A0), (A1) \)}

none = {}
univ = {\( (N0), (N1), (N2), (A0), (A1) \)}
iden = {\( (N0, N0), (N1, N1), (N2, N2), (A0, A0), (A1, A1) \)}
logic: set operators

+ union
& intersection
- difference
in subset
= equality

Name = {(N0), (N1), (N2)}
Alias = {(N1), (N2)}
Group = {(N0)}
RecentlyUsed = {(N0), (N2)}

Alias + Group = {(N0), (N1), (N2)}
Alias & RecentlyUsed = {(N2)}
Name - RecentlyUsed = {(N1)}
RecentlyUsed in Alias = false
RecentlyUsed in Name = true
Name = Group + Alias = true

greg = {(N0)}
rob = {(N1)}

greg + rob = {(N0), (N1)}
greg = rob = false
rob in none = false

cacheAddr = {(N0, A0), (N1, A1)}
diskAddr = {(N0, A0), (N1, A2)}
cacheAddr + diskAddr =
cacheAddr & diskAddr =
cacheAddr = diskAddr =
logic: product operator

$\rightarrow$ cross product

Name = \{(N0), (N1)\}
Addr = \{(A0), (A1)\}
Book = \{(B0)\}

Name$\rightarrow$Addr = \{(N0, A0), (N0, A1), (N1, A0), (N1, A1)\}
Book$\rightarrow$Name$\rightarrow$Addr =
\{(B0, N0, A0), (B0, N0, A1), (B0, N1, A0), (B0, N1, A1)\}

b = \{(B0)\}
b' = \{(B1)\}
address = \{(N0, A0), (N1, A1)\}
address' = \{(N2, A2)\}

b$\rightarrow$b' =

b$\rightarrow$address + b'$\rightarrow$address' =
logic: relational join

\[ p \cdot q \equiv (a, b) \cdot (a, d, c) = (a, a, d) \]

\[ x. f \equiv (c) \cdot (a, b) = (a) \]
logic: join operators

\[
. \quad \text{dot join} \quad e_1[e_2] = e_2.e_1 \\
[] \quad \text{box join} \quad a.b.c[d] = d.(a.b.c)
\]

- Book = \{(B0)\}
- Name = \{(N0), (N1), (N2)\}
- Addr = \{(A0), (A1), (A2)\}
- Host = \{(H0), (H1)\}

- myName = \{(N1)\}
- myAddr = \{(A0)\}

- address = \{(B0, N0, A0), (B0, N1, A0), (B0, N2, A2)\}
- host = \{(A0, H0), (A1, H1), (A2, H1)\}

- Book.address = \{(N0, A0), (N1, A0), (N2, A2)\}
- Book.address[myName] = \{(A0)\}
- Book.address.myName = \{}

- host[myAddr] = \{}
- address.host = \{}
logic: unary operators

\[ \neg \quad \text{transpose} \]
\[ \wedge \quad \text{transitive closure} \]
\[ \ast \quad \text{reflexive transitive closure} \]

apply only to binary relations

\[ \neg \text{next} = \{(N_1, N_0), (N_2, N_1), (N_3, N_2)\} \]
\[ \wedge \text{next} = \{(N_0, N_1), (N_0, N_2), (N_0, N_3), \]
\[ \quad (N_1, N_2), (N_1, N_3), \]
\[ \quad (N_2, N_3)\} \]
\[ \ast \text{next} = \{(N_0, N_0), (N_0, N_1), (N_0, N_2), (N_0, N_3), \]
\[ \quad (N_1, N_1), (N_1, N_2), (N_1, N_3), \]
\[ \quad (N_2, N_2), (N_2, N_3), (N_3, N_3)\} \]

\[ \wedge r = r + r.r + r.r.r + \ldots \]
\[ \ast r = \text{id} + \wedge r \]

\[
\text{Node} = \{(N_0), (N_1), (N_2), (N_3)\} \\
\text{next} = \{(N_0, N_1), (N_1, N_2), (N_2, N_3)\} \\
\neg \text{next} = \{(N_1, N_0), (N_2, N_1), (N_3, N_2)\} \\
\wedge \text{next} = \{(N_0, N_1), (N_0, N_2), (N_0, N_3), \}
\[ \quad (N_1, N_2), (N_1, N_3), \]
\[ \quad (N_2, N_3)\} \\
\ast \text{next} = \{(N_0, N_0), (N_0, N_1), (N_0, N_2), (N_0, N_3), \]
\[ \quad (N_1, N_1), (N_1, N_2), (N_1, N_3), \]
\[ \quad (N_2, N_2), (N_2, N_3), (N_3, N_3)\} \\
\text{first} = \{(N_0)\} \\
\text{rest} = \{(N_1), (N_2), (N_3)\} \\
\text{first.}^\wedge \text{next} = \text{rest} \\
\text{first.}^\ast \text{next} = \text{Node} \]
logic: restriction and override

\[
<: \quad \text{domain restriction} \\
>: \quad \text{range restriction} \\
++ \quad \text{override}
\]

\[
p ++ q = p - (\text{domain}[q] <: p) + q
\]

Name = \{(N0), (N1), (N2)\}
Alias = \{(N0), (N1)\}
Addr = \{(A0)\}
address = \{(N0, N1), (N1, N2), (N2, A0)\}

address :> Addr = \{(N2, A0)\}
Alias <: address = address :> Name = \{(N0, N1), (N1, N2)\}
address :> Alias = \{(N0, N1)\}

workAddress = \{(N0, N1), (N1, A0)\}
address ++ workAddress = \{(N0, N1), (N1, A0), (N2, A0)\}

\[
m' = m ++ (k \rightarrow v)
\]

update map m with key-value pair (k, v)
logic: boolean operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>not</td>
<td>negation</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
<td>conjunction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=&gt;</td>
<td>implies</td>
<td>implication</td>
</tr>
<tr>
<td>else</td>
<td>alternative</td>
<td></td>
</tr>
<tr>
<td>&lt;=&gt;</td>
<td>iff</td>
<td>bi-implication</td>
</tr>
</tbody>
</table>

four equivalent constraints:

\[
F \implies G \text{ else } H
\]

\[
F \text{ implies } G \text{ else } H
\]

\[
(F \&\& G) \lor ((\neg F) \&\& H)
\]

\[
(F \text{ and } G) \lor ((\neg F) \text{ and } H)
\]
## logic: quantifiers

<table>
<thead>
<tr>
<th>Quantifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all x: e</td>
<td>F holds for every x in e</td>
</tr>
<tr>
<td>all x: e1, y: e2</td>
<td>F holds for at least one x in e</td>
</tr>
<tr>
<td>all x, y: e</td>
<td>F holds for no x in e</td>
</tr>
<tr>
<td>all disj x, y: e</td>
<td>F holds for at most one x in e</td>
</tr>
<tr>
<td>one x: e</td>
<td>F holds for exactly one x in e</td>
</tr>
</tbody>
</table>

### Examples

- **some**
  - `some n: Name, a: Address | a in n.address`
  - some name maps to some address — address book not empty

- **no**
  - `no n: Name | n in n.^address`
  - no name can be reached by lookups from itself — address book acyclic

- **all**
  - `all n: Name | lone a: Address | a in n.address`
  - every name maps to at most one address — address book is functional

- **all**
  - `all n: Name | no disj a, a': Address | (a + a') in n.address`
  - no name maps to two or more distinct addresses — same as above
logic: set declarations

set declarations

\[ \text{x: m e} \quad \text{Q x: m e} \quad \text{x: e} \leftrightarrow \text{x: one e} \]

- set: any number
- one: exactly one
- lone: zero or one
- some: one or more

RecentlyUsed: set Name

RecentlyUsed is a subset of the set Name

senderAddress: Addr

senderAddress is a singleton subset of Addr

senderName: lone Name

senderName is either empty or a singleton subset of Name

receiverAddresses: some Addr

receiverAddresses is a nonempty subset of Addr
logic: relation declarations

\[
\begin{align*}
  & r: A \ x \rightarrow n \ B \\
  & Q \ r: A \ y \rightarrow n \ B \\
  & r: A \rightarrow B \iff \\
  & r: A \set \rightarrow \set B
\end{align*}
\]

\[
(\ r: A \ x \rightarrow n \ B \ ) \iff \\
((\all a: A \ | \ n \ a.r) \ \text{and} \ (\all b: B \ | \ m \ r.b))
\]

workAddress: Name \rightarrow \text{one} \ Addr
each alias refers to at most one work address

homeAddress: Name \rightarrow \text{one} \ Addr
each alias refers to exactly one home address

members: Name \text{one} \rightarrow \text{some} \ Addr
address belongs to at most one group name
and group contains at least one address

\[
\begin{align*}
  & r: A \rightarrow (B \ y \rightarrow n \ C) \iff \\
  & \text{all} \ a: A \ | \ a.r: B \ x \rightarrow n \ C \\
  & r: (A \ x \rightarrow n \ B) \rightarrow C \iff \\
  & \text{all} \ c: C \ | \ r.c: A \ y \rightarrow n \ B
\end{align*}
\]
logic: quantified expressions

| some e | e has at least one tuple |
| no e   | e has no tuples          |
| lone e | e has at most one tuple  |
| one e  | e has exactly one tuple  |

some Name
set of names is not empty

some address
address book is not empty – it has a tuple

no (address.Addr - Name)
nothing is mapped to addresses except names

all n: Name | lone n.address
every name maps to at most one address
logic: comprehensions

\{x_1: e_1, x_2: e_2, \ldots, x_n: e_n \mid F\}

\{n: \text{Name} \mid \text{no } n.\text{^address} & \text{Addr}\}
set of names that don't resolve to any actual addresses

\{n: \text{Name}, a: \text{Address} \mid n \rightarrow a \text{ in } \text{^address}\}
binary relation mapping names to reachable addresses
logic: if and let

\[
\begin{align*}
& f \implies e_1 \text{ else } e_2 \\
& \text{let } x = e \mid \text{ formula} \\
& \text{let } x = e \mid \text{ expression}
\end{align*}
\]

four equivalent constraints:

\[
\begin{align*}
\text{all } n: \text{Name} \mid \\
& (\text{some } n.\text{workAddress} \\
& \implies n.\text{address} = n.\text{workAddress} \\
& \text{else } n.\text{address} = n.\text{homeAddress})
\end{align*}
\]

\[
\begin{align*}
\text{all } n: \text{Name} \mid \\
& \text{let } w = n.\text{workAddress}, a = n.\text{address} \mid \\
& (\text{some } w \implies a = w \text{ else } a = n.\text{homeAddress})
\end{align*}
\]

\[
\begin{align*}
\text{all } n: \text{Name} \mid \\
& \text{let } w = n.\text{workAddress} \mid \\
& n.\text{address} = (\text{some } w \implies w \text{ else } n.\text{homeAddress})
\end{align*}
\]

\[
\begin{align*}
\text{all } n: \text{Name} \mid \\
& n.\text{address} = (\text{let } w = n.\text{workAddress} \mid \\
& (\text{some } w \implies w \text{ else } n.\text{homeAddress}))
\end{align*}
\]
logic: cardinalities

\[
\#r \quad \text{number of tuples in } r \\
0, 1, \ldots \quad \text{integer literal} \\
+ \quad \text{plus} \\
- \quad \text{minus}
\]

\[
= \quad \text{equals} \\
< \quad \text{less than} \\
> \quad \text{greater than} \\
\leq \quad \text{less than or equal to} \\
\geq \quad \text{greater than or equal to}
\]

**sum** \( x: e \mid ie \)

*sum of integer expression* \( ie \) *for all singletons* \( x \) *drawn from* \( e \)

\[
\text{all } b: \text{Bag} \mid \#b.\text{marbles} \leq 3 \\
\text{all bags have 3 or less marbles}
\]

\[
\#\text{Marble} = \text{sum } b: \text{Bag} \mid \#b.\text{marbles} \\
\text{the sum of the marbles across all bags} \\
\text{equals the total number of marbles}
\]
2 logics in one

• “everybody loves a winner”

• predicate logic
  - $\forall w \ | \ \text{Winner}(w) \Rightarrow \forall p \ | \ \text{Loves}(p, w)$

• relational calculus
  - $\text{Person} \times \text{Winner} \subseteq \text{loves}$

• Alloy logic – any way you want
  - all $p: \text{Person}, w: \text{Winner}$ | $p \rightarrow w$ in loves
  - $\text{Person} \rightarrow \text{Winner}$ in loves
  - all $p: \text{Person}$ | Winner in $p$.loves
logic exercises: binary relations & join

➢ Download *properties.als from the tutorial website*
  – explores properties of binary relations

➢ Download *distribution.als from the tutorial website*
  – explores the distributivity of the join operator

➢ Follow the instructions in the models

➢ Don't hesitate to ask questions
logic exercise: modeling the tube

➢ Download *tube.als* from the tutorial website

• a simplified portion of the London Underground:

➢ follow the instructions in the model